Praxisorientierte Einführung in C++
Lektion: "Mehrfachvererbung"

Christof Elbrechter

Neuroinformatics Group, CITEC

June 12, 2014

Table of Contents

Motivation

- Bis jetzt: Klasse hat keine oder eine Oberklasse
- C++ erlaubt auch, mehr als eine Klasse/Struktur zu beerben
- Syntax ähnlich wie Initialisierungsliste

```
struct EierlegendeWollmilchSau : public Huhn, public Rind,
                                 public Schwein, public Schaf
      // wichtig: Vererbungs-Qualifier binded nur an das naechste Token
{

};
```

Motivation

- Wir modellieren Eigenschaften als Klassenhierarchien
- Beispiel:

- Apparat
  - Motorgetrieben
  - Windgetrieben
- Organismus
  - Pflanze
  - Tier

- Klasse `Auto` kann z.B. `Motorgetrieben` beerben
- Klasse `Mensch` beerbt Klasse `Tier`
- Aber was ist mit einer Klasse `Cyborg`?
  - Ein `Cyborg` ist ein `Apparat` und ein `Organismus`

Motivation

- Lösung: Klasse `Cyborg` erbt von Apparat und Organismus

Mehrfachvererbung

```
struct Organism {
  virtual ~Organism() { }
  virtual void live() = 0;
};
```

```
struct Apparatus {
  virtual ~Apparatus() { }
  virtual void operate() = 0;
};
```

```
struct Cyborg : public Organism, public Apparatus {
  virtual void live() {
    // do the cyborg organism thang
  }
  virtual void operate() {
    // do the cyborg apparatus thang
  }
};
```

Mehrfachvererbung

- Klasse `Cyborg` gehorcht jetzt den Schnittstellen von `Apparatus` und `Organism`
- Polymorphismus funktioniert mit Referenzen/Zeigern auf `Apparatus` und `Organism`
- Dabei kann immer nur auf *einen* Aspekt des Objekts zugegriffen werden...

Mehrfachvererbung

- Beispiel: Zugriff über `Organism`- oder `Apparatus`-Referenz

```
void age(Organism &o) {
  o.live() // Zugriff auf Organism-Aspekte von o
}

void wear(Apparatus &a) {
  a.operate() // Zugriff auf Apparatus-Aspekte von a
}

int main() {
  Cyborg c;
  age(c);
  wear(c);
}
```

Namensauflösung

- Was ist, wenn Basisklassen gemeinsamen Bezeichner haben?
- Ausdrücke sind nicht mehr eindeutig

```
struct Organism {
       string name;
       float getAge() const { ... }
};

struct Apparatus {
       string name;
       float getAge() const { ... }
};

struct Cyborg : public Organism, public Apparatus { };

int main() {
  Cyborg c;
  c.name = "Christof"; // Fehler: nicht eindeutig
  c.getAge();          // ditto
}
```

Namensauflösung

- Lösung, die "immer funktioniert":
  - In Ausdrücken: Explizite Nennung des "Slices"
  - Basisklassen-Präfix vor Bezeichner bei Aufruf der Methode/Benutzung der Elementvariable im Ausdruck...

```
struct Organism { string name; float getAge(){} };

struct Apparatus { string name; float getAge(){} };

struct Cyborg : public Organism, public Apparatus { };

int main() {
  Cyborg c;
  c.Organism::name = "Christof";
  float age = c.Apparatus::getAge();
}
```

Namesauflösung - Alternative

- Bei Methoden Alternative: Explizites Überschreiben oder "Importieren" von Methoden aus Basisklasse...
- ...in vielen Fällen muss dann aber was anderes passieren:
  - Reimplementation der entsprechenden Funktion

```
struct Organism { string name; float getAge(){} };

struct Apparatus { string name; float getAge(){} };

struct Cyborg : public Organism, public Apparatus {
  Organism::getAge;
};

int main() {
  Cyborg c;
  float age = c.getAge(); // eindeutig
}
```

Namensauflösung - Elementvariablen

- Leider funktioniert das nicht mit Elementvariablen
- D.h.: Elemente in Basisklassen `protected` (oder `private`) machen und Zugriff über (virtuelle) Methoden regeln
- Entspricht auch eher dem Prinzip des "Information Hiding"

Mehrfachvererbung - Anmerkung

- Gleiche Bezeichner in Basisklassen manchmal auch Hinweis auf mögliche Modellierung mit gemeinsamer Basisklasse
  - Diamantförmiger Vererbungsgraph
  - "Virtuelle" Vererbung (später)

Konstruktoren

- Bei Einfachvererbung:
  - Konstruktor der Basisklasse wird vor Konstruktor der abgeleiteten Klasse aufgerufen (rekursiv)
  - Welcher? Initialisierungsliste!
- Bei Mehrfachvererbung:
  - Konstruktoren der Basisklassen werden vor Konstruktor der abgeleiteten Klasse aufgerufen
  - Reihenfolge, wie in der Vererbungsliste
  - Initialisierungsliste des Konstruktors kann spezielle Konstruktoraufrufe für Basisklassen enthalten

Konstruktoren - Beispiel

```
struct Organism {
  Organism(const DNA &dna) { ... }
  Organism() { ... }
};

struct Apparatus {
  Apparatus(const Schematic &schematic) { ... }
  Apparatus() { ... }
};

struct Cyborg : public Organism, public Apparatus {
  Cyborg() { ... }
  Cyborg(const DNA &dna) : Organism(d) { ... }
  Cyborg(const Schematic &s) : Apparatus(s)
  { ... }
  Cyborg(const Schematic &s, const DNA &d) :
    Apparatus(s), Organism(d)
  { ... }
};
```

Destruktoren

- Destruktoren werden in umgekehrter Reihenfolge aufgerufen (umgekehrt zu der Reihenfolge in der Vererbungsliste)

Virtuelle Vererbung - Motivation

- Beispiel: Modellierung von Fahrzeugtypen
- Gemeinsame Oberklasse `Fahrzeug`
- Einteilung in
  - `Landfahrzeug`
  - `Wasserfahrzeug`
  - `Luftfahrzeug`

Virtuelle Vererbung - Motivation

- Ein spezielles Fahrzeug beerbt eine dieser Klassen
- Beispiele für abgeleitete Klassen
  - `Auto`
  - `Fahrrad`
  - `Motorboot`
  - `Segelboot`
  - `Segelflugzeug`
  - etc..
- Aber was ist mit einem Amphibien-fahrzeug?
  - Amphibienfahrzeug erbt von `Landfahrzeug` und `Wasserfahrzeug`

Virtuelle Vererbung - Motivation - Beispiel

```
struct Fahrzeug {
  float m_posX, m_posY;
  virtual void fahrNach(float x, float y) {
    m_posX = x; m_posY = y;
  }
  virtual float maxGeschwindigkeit() = 0;
  virtual ~Fahrzeug() { }
};

struct LandFahrzeug : public Fahrzeug {
  virtual float maxSteigung() = 0;
};

struct WasserFahrzeug : public Fahrzeug {
  virtual float maxWindStaerke() = 0;
  virtual float minWasserTiefe() = 0;
};

struct LuftFahrzeug : public Fahrzeug {
  virtual float maxHoehe() = 0;
};
```

Virtuelle Vererbung - Motivation - Beispiel

- Eine spezielle Klasse, z.B. `Maserati` erbt von einer dieser Klassen, und implementiert ggf. Methoden

```
struct Maserati : public Landfahrzeug {
  virtual float maxSteigung() { return 30; }
  virtual float maxGeschwindigkeit() { return 210; }
};
```

- Bis hierhin ist alles gut :D

Virtuelle Vererbung - Motivation

- VW Typ 166 ist ein Amphibienfahrzeug, ist also Land- und Wasserfahrzeug
- Klasse `VWTyp166` erbt also von beiden Klassen

- Quelle: `http://de.wikipedia.org/wiki/Bild:VW_Schwimmwagen_1.jpg`

Virtuelle Vererbung - Motivation

```
struct VWTyp166 :
  public LandFahrzeug,
  public WasserFahrzeug
{
  virtual float maxSteigung() { return 20; }
  virtual float maxGeschwindigkeit() { return 100; }
  virtual float maxWindStaerke() { return 6.5; }
  virtual float minWasserTiefe() { return 0; }
};
```

- Klasse `VWTyp166` erbt somit alle Methoden und Elemente aller Elternklassen
- Soweit, so gut, aber was ist mit `m_posX` und `m_posY`? Oder mit `fahrNach()`
  - Die gibt es jetzt zweimal

```
VWTyp166 superTeil;
superTeil.fahrNach(0,0); // Fehler. Aufruf ist mehrdeutig :(
```

Virtuelle Vererbung - Explizite Nennung

- Explizites Präfix auch hier möglich

```
VWTyp166 superTeil;
superTeil.LandFahrzeug::fahrNach(0,0);
superTeil.WasserFahrzeug::fahrNach(1,1);
```

- Allerdings nicht immer sinnvoll
  - Gespaltene Persönlichkeit!

Virtuelle Vererbung - Die Lösung

- Lösung: virtuelle Vererbung
- Schlüsselwort `virtual` (noch mal)
- Bewirkt, dass in abgeleiteten Klassen die virtuell beerbte Klasse nur einmal vorkommt
- Im Beispiel: `LuftFahrzeug`, `WasserFahrzeug` und `LandFahrzeug` sollten `Fahrzeug` virtuell beerben

Virtuelle Vererbung - Beispiel

```
struct Fahrzeug {
  float m_posX, m_posY;
  virtual void fahrNach(float x, float y) {
    m_posX = x; m_posY = y;
  }
  virtual float maxGeschwindigkeit() = 0;
  virtual ~Fahrzeug() { }
};

struct LandFahrzeug : public virtual Fahrzeug {
  virtual float maxSteigung() = 0;
};

struct WasserFahrzeug : public virtual Fahrzeug {
  virtual float maxWindStaerke() = 0;
  virtual float minWasserTiefe() = 0;
};

struct LuftFahrzeug : public virtual Fahrzeug {
  virtual float maxHoehe() = 0;
};
```

Virtuelle Vererbung - Beispiel

- Jetzt geht's

```
struct VWTyp166 :
  public LandFahrzeug,
  public WasserFahrzeug
{
  virtual float maxSteigung() { return 20; }
  virtual float maxGeschwindigkeit() { return 100; }
  virtual float maxWindStaerke() { return 6.5; }
  virtual float minWasserTiefe() { return 0; }
};

VWTyp166 superTeil;
superTeil.fahrNach(0,0);
```

Warum so kompliziert?

- Warum so kompliziert?
- Es macht *manchmal* auch Sinn, Vererbung für Aggregation zu nutzen. Dann braucht man evtl. zwei identische Slices der gemeinsamen Basisklasse
- Faustregel: Wenn Mehrfachvererbung für Modellierung von "ist-ein"-Beziehungen möglich sein soll: virtuelle Vererbung von allen gemeinsamen Basisklassen

Aggregation oder Vererbung?

- Manchmal macht Aggregation mehr Sinn als Mehrfachvererbung
- `Fahrzeug` (`LuftFahrzeug`, `LandFahrzeug`, `Wasserfahrzeug`)
- `Maschine` (`Luftgetrieben`, `Oelgetrieben`, `Muskelkraftgetrieben`)
- Alle Kombinationen explizit als Klassen modellieren: 3 * 3 = 9 Möglichkeiten

Aggregation oder Vererbung

- Alternative: Durch Aggregation modellieren:

```
struct Fahrzeug {
    Maschine *m_maschine;
};

struct WasserFahrzeug : public Fahrzeug {
    ...
};

int main() {
    WasserFahrzeug ruderBoot;
    ruderBoot.m_maschine = new MuskelkraftMaschine();
}
```

- Dadurch nur noch $3 + 3 = 6$ Klassen explizit zu modellieren
- Verlust von feiner Kontrolle über Verhalten
  - In diesem Fall aber vielleicht näher an der menschlichen Wahrnehmung